Minneshantering

Kapitel 7 (sid. 303-322)

Detta är det första kapitlet om minneshantering. Kapitel 7 introducerar grundläggande begrepp utifrån en enklare minneshanteringsmodell (utan virtuellt minne). D v s det minne som hanteras är begränsat till det minne som rent fysiskt är installerat.

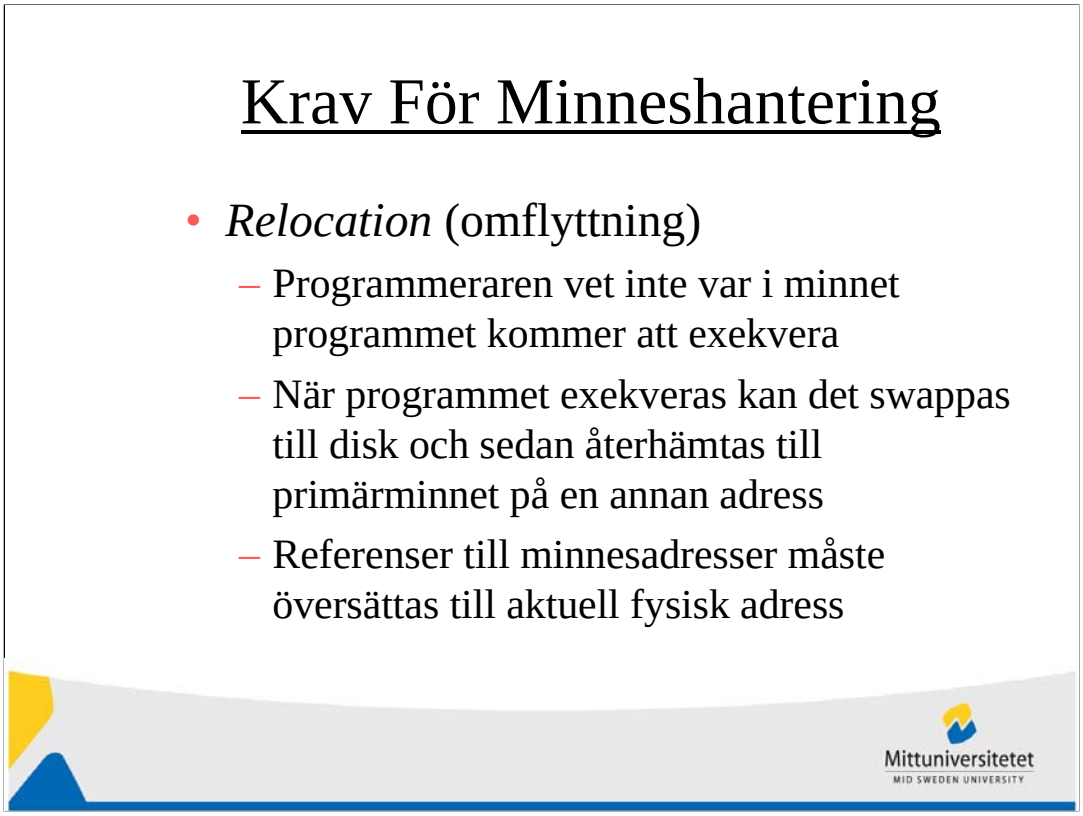
Memory management system (MMS)

Bild 1: Vid uniprogrammering, d v s endast ett program laddat i taget, använder kerneln och program var sin del av minnet. Vid multiprogrammering ska flera program dela på användardelen av minnet. Detta hanteras av OS via MMS på ett dynamiskt sätt. Målet är att ladda så många program som möjligt i minnet, så att processorn hela tiden har tillgång till processer som är klara att exekveras.

Minneshantering

- Dela upp minnet i mindre delar, en för varje process
- Minnet måste allokeras effektivt för att tillåta så många samtidiga processer i minnet som möjligt

Mittuniversitetet
MID SWEDEN UNIVERSITY



Krav på MMS

Relocation

Bild 3: Eftersom det kan finnas flera program som delar på minnet kan ett enskilt program inte garanteras ett specifikt minnesområde när det läses in i minnet. Processerna kommer dessutom att flyttas fram och tillbaka mellan primär och sekundärminnet under tiden som MMS försöker optimera minneshanteringen och klämma in fler processer. När en process flyttas tillbaka in i primärminnet, för att exekvera, så kan det bli på en annan plats än tidigare. OS måste därför hantera en översättning av adressreferenser i programmet till en aktuell fysisk adress i minnet.

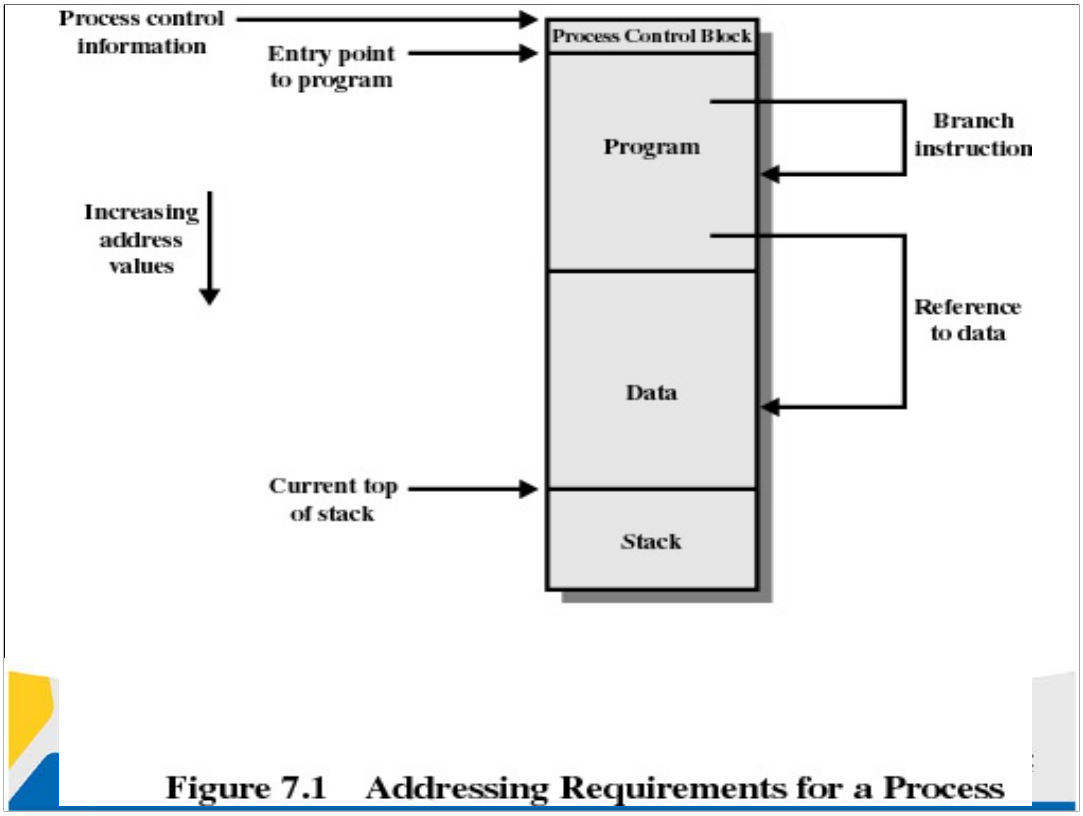


Figure 7.1 Addressing Requirements for a Process

Bild 4: visar en *processimage* (den process som exekveringen av programmet skapar dynamiskt) och viktiga minnesreferenser för en process. I det här fallet är programmet delat i sektioner, en för kod och en för data.

OS behöver skriva och läsa i processens kontrollblock och hitta var programmet börjar (entry point). En branch-instruktion är ju ett hopp inom programkoden medan en datareferens sker till datasektionen. Viktigt är också pekaren till stackens topp.

(På bilden ser det ut som om stacken är helt fylld, eftersom pekaren ligger i gränsen mot dataområdet. Stacken har ett minnesutrymme allokerat som den kan växa i.)

Krav På Minneshantering

- *Protection* (dataskydd)
  - Processer ska inte kunna referera till minne som hör till andra processer, utan tillstånd
  - Program kan inte hantera absoluta minnesadresser eftersom det kan relokeras
  - Under exekveringen måste alla minnesreferenser kontrolleras
    - Operativsystem kan inte hantera alla minnesreferenser som ett program gör

Mittuniversitetet
MID SWEDEN UNIVERSITY

Protection

Bild 5: En process ska inte, utan tillstånd, kunna referera (läsa/skriva) till en annan process minnesområde och ingen process får referera till kernelns eget minnesområde.

P g a relocation blir detta extra svårt att hantera. Varje process ”äger” ju inte ett visst minnesintervall hela tiden. Dessutom är ju exekveringen av ett program en dynamisk process, så kontrollen av referenser måste ske vid exekveringen av varje instruktion. OS ”hinner” inte hantera denna kontroll, utan det måste finnas stöd i hårdvaran för detta (mer om det senare).

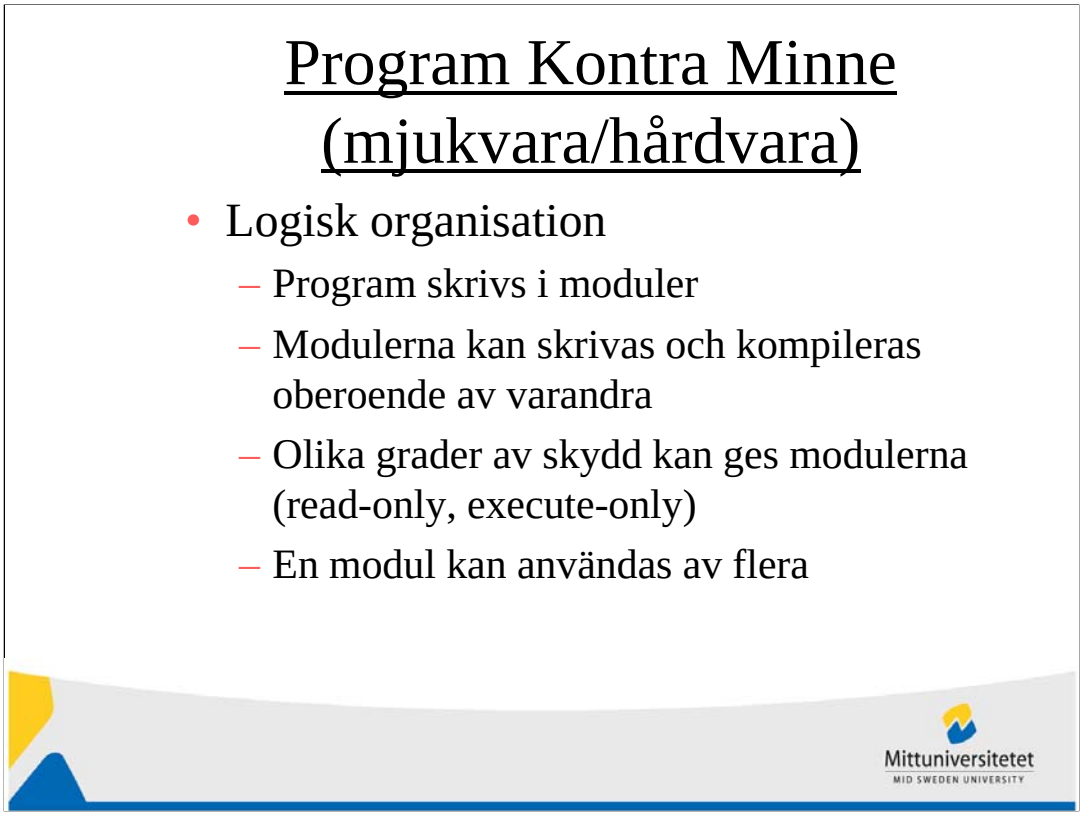
Krav På Minneshantering

- *Sharing*
  - Tillåter flera processer att dela samma del av minnet
  - Effektivare att låta flera processer ha tillgång samma data(adress) än att varje ska ha en egen kopia av data.

Mittuniversitetet
MID SWEDEN UNIVERSITY

Sharing

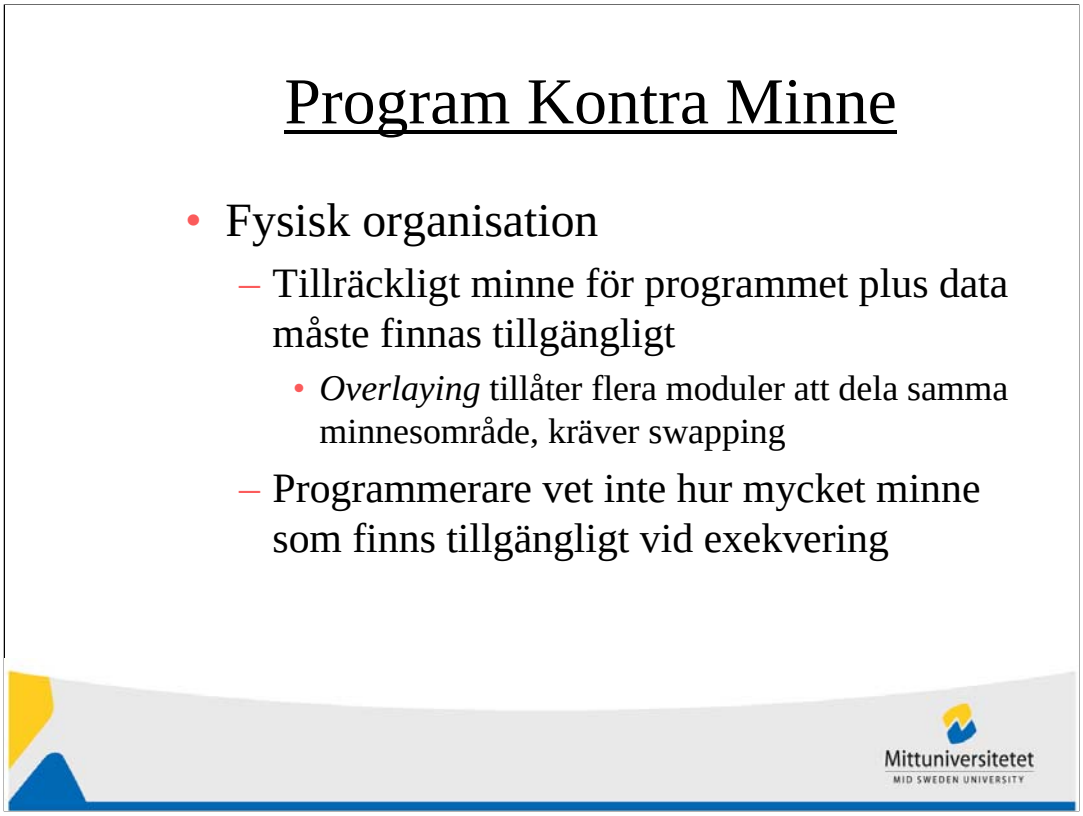
Bild 6: Data ska skyddas, men ändå ska det finnas möjlighet till kontrollerad delning av data och även kod.



Logisk organisation av minnet

Bild 7: Program skrivs oftast i form av moduler, oftast ur objektorienterad synpunkt. D v s vilken funktion de har. Modulerna ska kunna ha olika former av skydd (read/write/execute etc.) och kunna delas mellan flera olika processer.

Denna moduluppdelning på logisk nivå återspeglas inte direkt i det fysiska minnet, som ju bara består av ett linjärt adressområde. MMS måste därför hantera denna modulisering



Fysisk organisation av minnet

Bild 8: Det fysiska minnet består däremot oftast av två nivåer (tre om cache), i termer av primär och sekundärminne. OS hanterar flyttningen av data mellan dessa och ser till att det finns tillräckligt med minne för kod och data. Kom ihåg det primära syftet: att hålla processorn fullt sysselsatt. OS kan pressa in fler processer än vad som det egentligen finns fysiskt minne till genom att använda *overlaying*. Genom att flytta bitar av inte för ögonblicket exekverande processer till sekundärminnet så kan flera processer få plats samtidigt (swapping). Men, eftersom vi ännu inte har introducerat virtuellt minne så kan en process inte referera mer minne än vad som fysiskt finns.

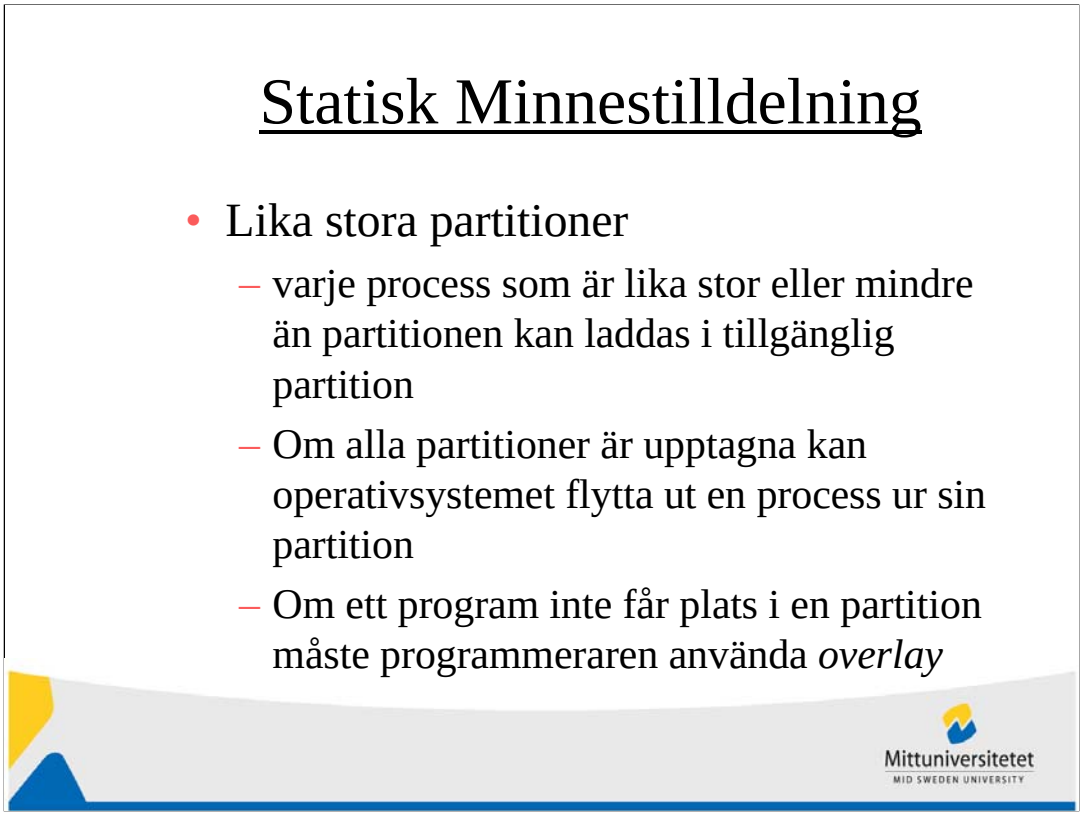
Partitionering av minnet

Virtuellt minne, som behandlas i nästa kapitel, bygger på två huvudprinciper för att dela upp minnet i mindre delar:

• • *Paging*, minnet delas i små delar av fast storlek, som är anpassade till sekundärminnets organisation

• • *Segmentation*, uppdelning utifrån programmets olika delar kod/data

Innan vi kommer dit ska vi titta på en äldre teknik, minnespartitionering. Vi ska först titta på grundläggande tekniker för att: bestämma storlek på partitionerna och hur vi ska fylla dessa med önskat data.



Statisk partitionering (fast storlek)

Vänta med att titta på Table 7.1 på sidan 307. Den blir en bra sammanfattning senare.

Bild 9-11: Minnet är uppdelat i delar med fast storlek. Vi börjar med att de har samma storlek.

Grunden är att OS plockar in processer i partitionerna efterhand som det finns flera körklara. Om alla processer är avbrutna (suspended) plockar OS in fler processer för att hålla processorn sysselsatt. Om det då redan är fullt i minnet, så kan någon av de avbrutna processerna swappas ut, så att den nya får plats.

Det finns två problem med denna teknik:

• • Programmet får inte plats i en partition. Lösningen är att programmeraren använder overlay-teknik d v s skriver programmet i mindre moduler som kan laddas och köras var för sig.

• • Viktigast är dock att utnyttjandet av minnet blir dåligt. Även om en process inte fyller sin partition, så kan ingen annan kan använda det lediga utrymmet i alla fall. Detta kallas **intern fragmentering** (det är ju inne i partitionerna som utrymmet slösas).

Vi kan försöka lösa detta genom att ha partitioner med olika storlek och försöka plocka in processerna i en så liten som möjligt. Detta görs då med en placeringsalgoritm (*placement algoritm*).

Statisk Minnestilldelning

- På detta sätt används inte minnet effektivt. Ett program oavsett hur litet tar upp en hel partition. Detta kallas intern fragmentering (*internal fragmentation*)

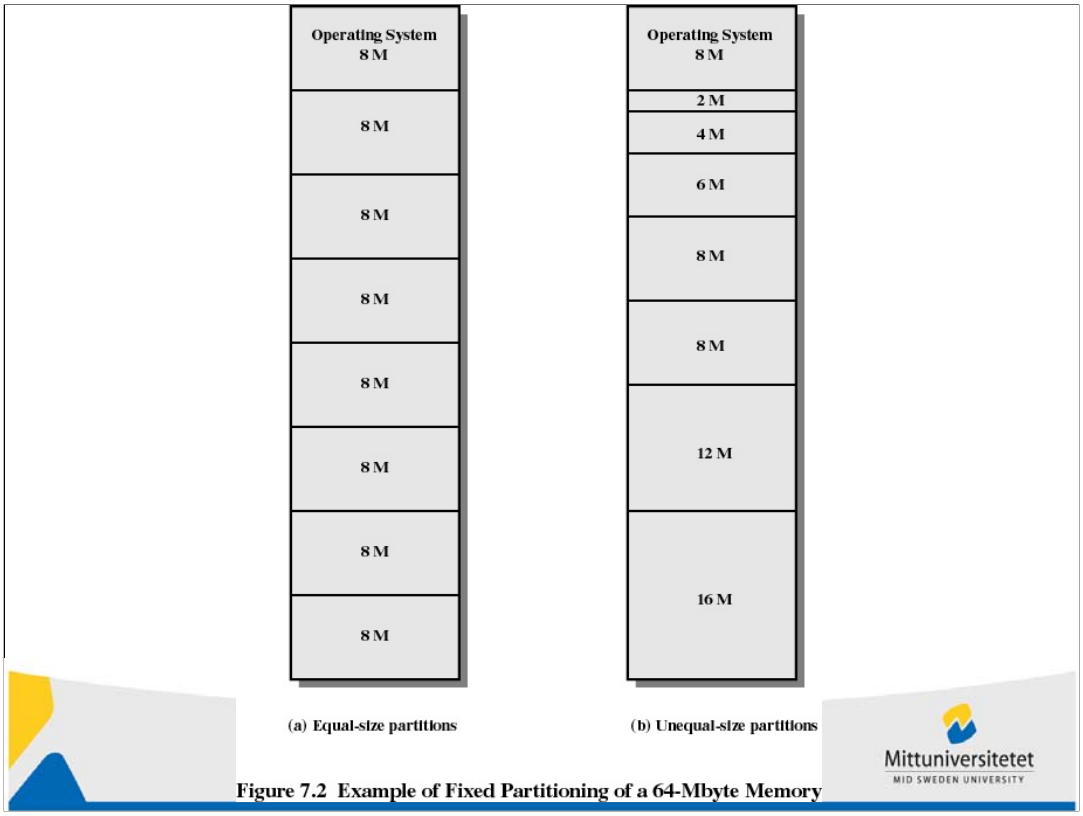


Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

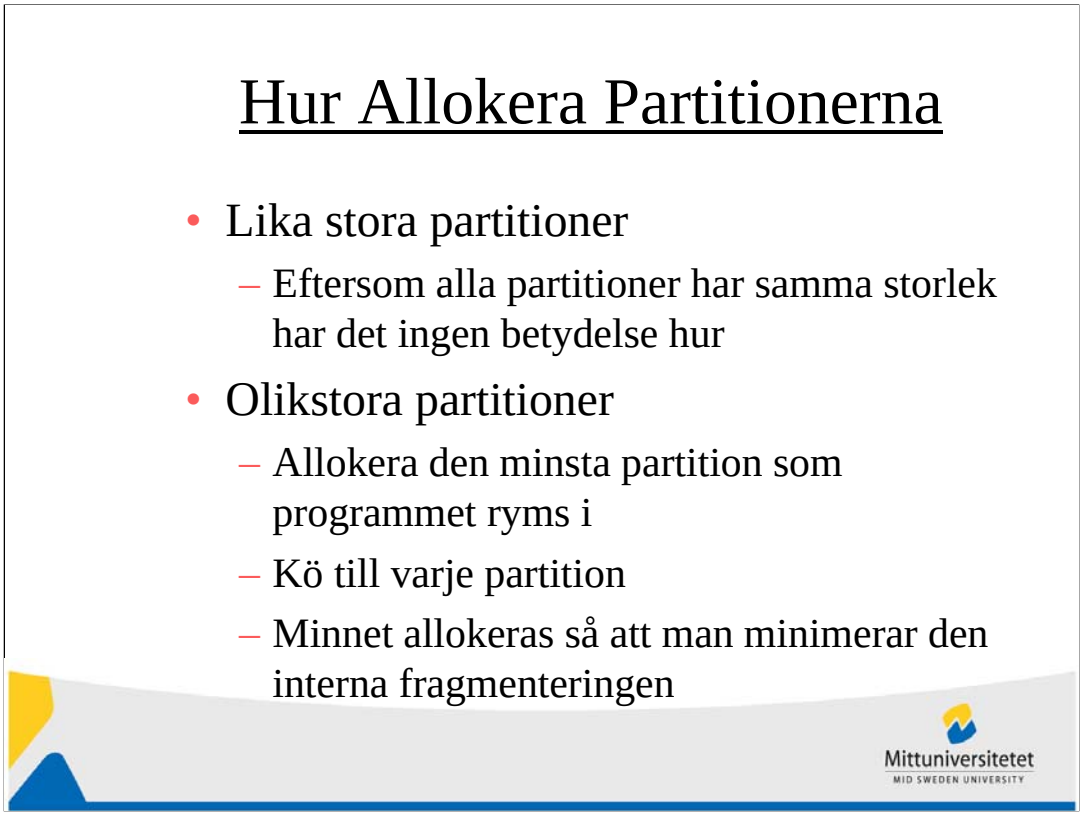


Bild 12-13: behandlar detta. Om vi har lika stora partitioner så har det ju ingen betydelse var vi placerar processen. Finns det bara en ledig så lägg den där. Är alla upptagna så måste vi bestämma vilken som ska swappas ut. Det kommer vi tillbaka till i kapitel 9.

Med olika stora partitioner är det enkelt att skapa en kö till varje partition, så att utswappade processer kan läggas i kön till rätt storlek. Vi har nu minimerat den interna fragmenteringen. Vi kan dock hamna i den situationen att vi har små processer som ska in, men bara stora partitioner lediga men då ligger de i fel kö och kommer inte in i större lediga.

Smartare är då att ha en enda kö. Processerna placeras nu i den minsta möjliga lediga partitionen. Finns ingen ledig är vi tillbaka i swapping.

Två nackdelar finns med att använd statiska partitioner, oavsett storlek:

• • Antalet partitioner, som bestäms när man skapar systemet, bestämmer det maximala antalet aktiva processer.

• • Partitionernas storlek är också bestämt från början och kan inte ändras om det kommer en massa mycket små jobb.

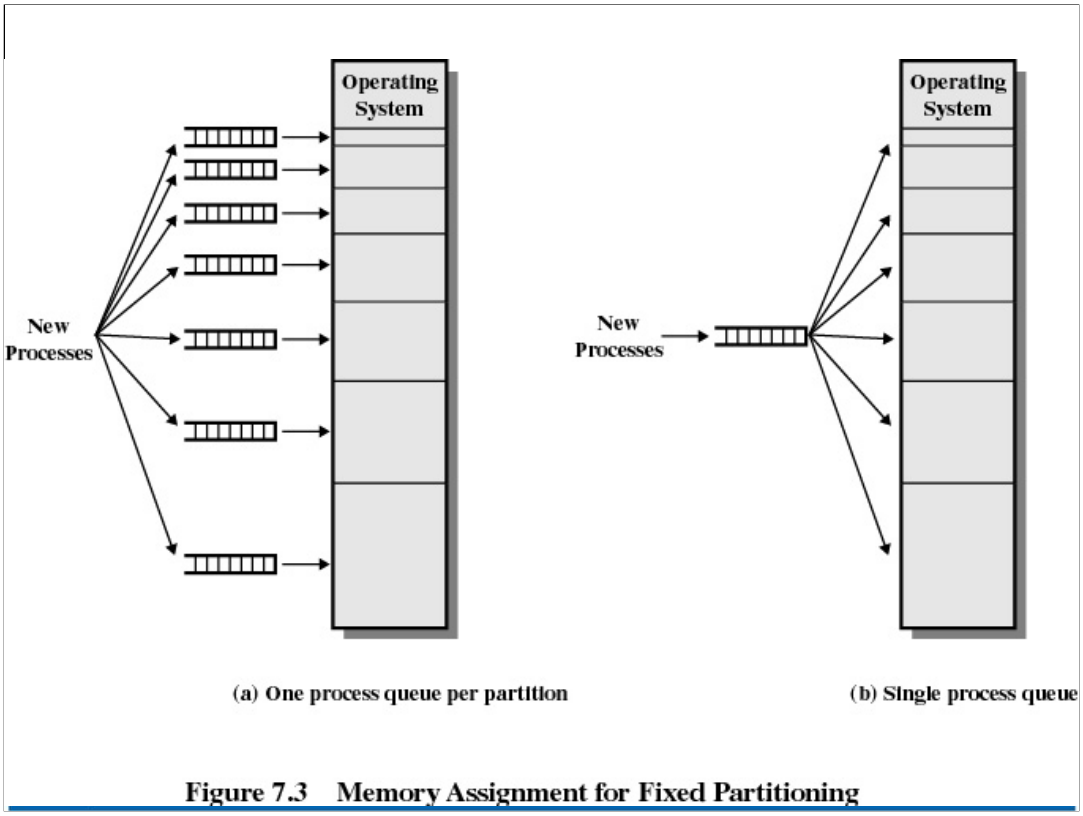
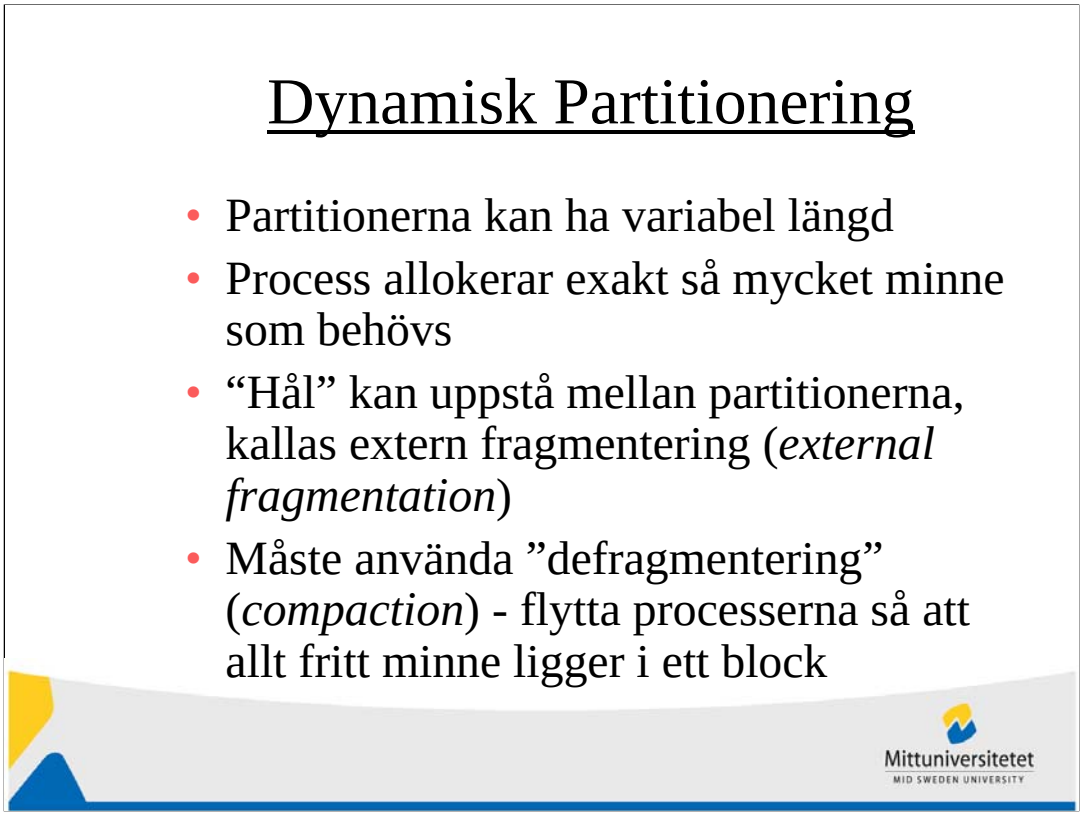


Figure 7.3 Memory Assignment for Fixed Partitioning



Dynamisk partitionering

Bild 14: Lösningen kanske är, att inte ha en fast storlek? Processen tilldelas så mycket minne den vill ha.

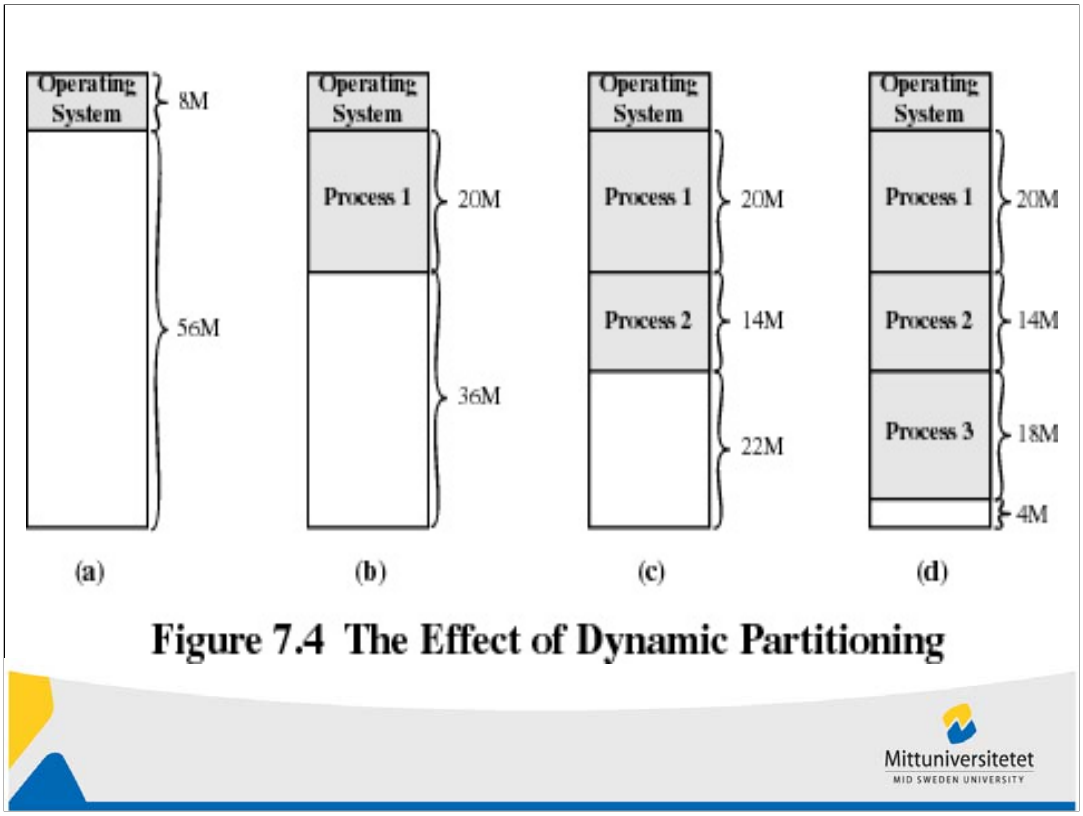
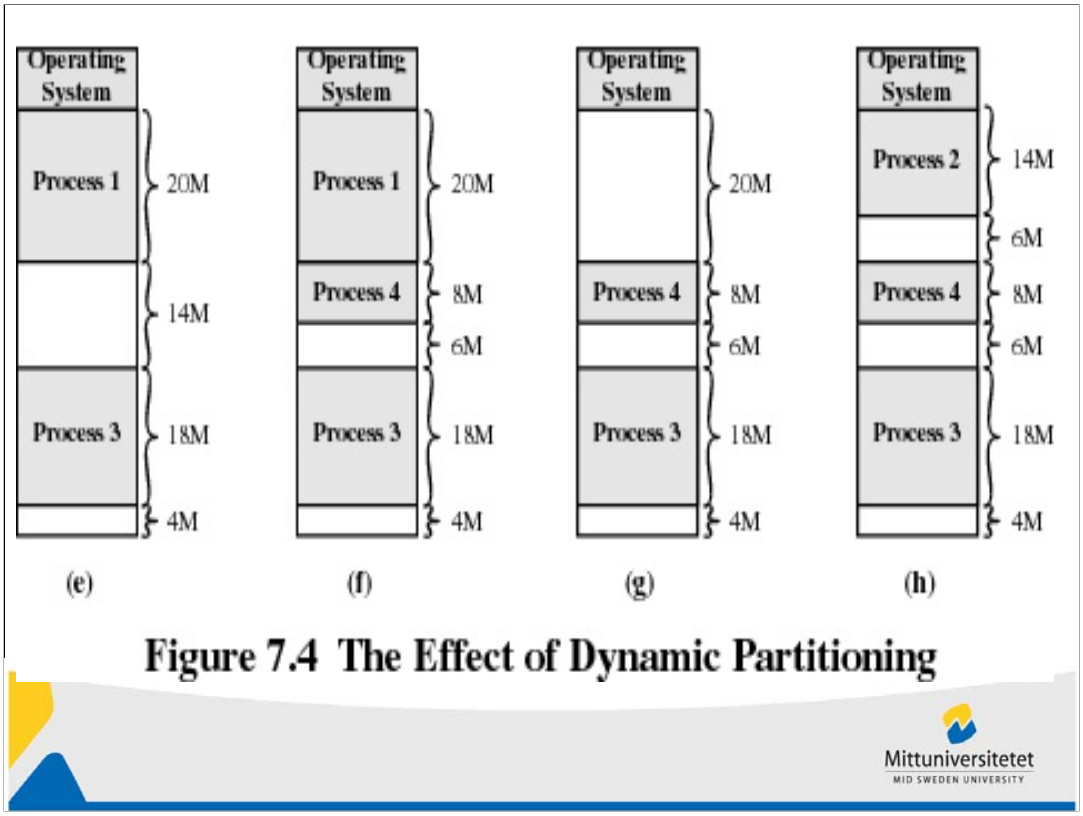
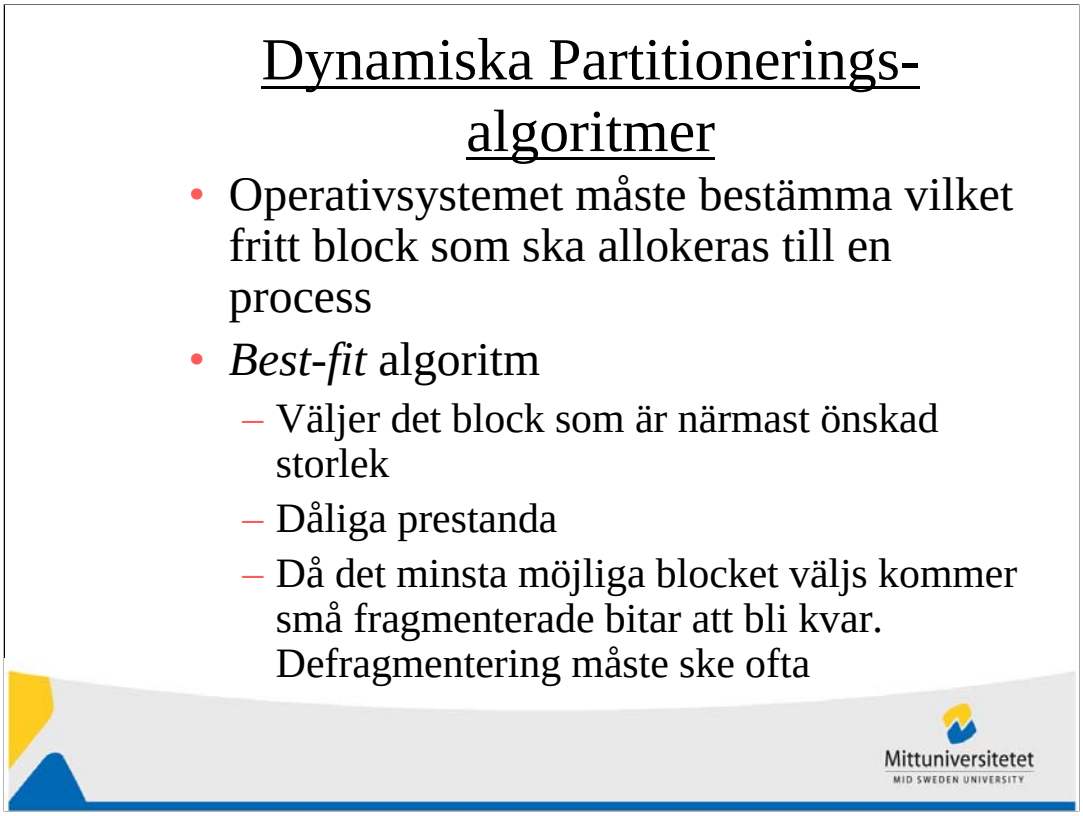


Bild 15: Visar hur snyggt det skulle kunna vara, men som vanligt blir det andra problem.

Sidan 311



och *Bild 16:* beskriver hur illa det kan gå. Nu får vi istället **extern fragmentering** (fragmenteringen är utanför partitionerna)**.** Små bitar av ledigt minne som inte går att fylla med de processer som väntar. Vi kan försöka lösa detta med *compaction* d v s med jämna mellanrum flyttas processerna om i minnet så att det lediga utrymmet samlas (som vanlig defragmentering av hårddisk). Detta tar dock tid och kräver att vi hanterar relocation.



Placement algorithm

Här hamnar vi i tre (fyra) placeringsalgoritmer:

• • Best-fit, *Bild 17:* väljer det utrymme som är närmast önskad storlek. Måste scanna hela minnet för att se var den lämpligaste ligger, vilket tar tid. Eftersom processens storlek kommer att vara nära partitionens så får vi en massa mycket små fragment som kräver defragmenterng

Dynamiska Partitionerings-algoritmer

- *First-fit* algoritm
  - Snabbast
  - Kan ha många processer laddade i början av minnet. Dessa måste först sökas igen för att hitta ett ledigt block

Mittuniversitetet
MID SWEDEN UNIVERSITY

• • First-fit, *Bild 18:* minskar söktiden genom att starta scanningen från början och välja första lämpliga utrymme. Nackdelen är att när vi börjar att ha många processer så är de flesta laddade i början av minnet och alla dessa måste sökas igenom innan vi hittar ett hål.

Dynamiska Partitionerings-algoritmer

- *Next-fit*
  - Kommer oftare att placera blocket i slutet av minnet där de största blocken finns
  - Det största minnesblocket kommer att splittrar upp i mindre bitar
  - Defragmentering måste användas för att få ett stort ledigt block i slutet av minnet

Mittuniversitetet
MID SWEDEN UNIVERSITY

• • Next-fit, *Bild 19:* Vi försöker hantera detta genom att söka från senast placerade process, tills vi hittar en plats.

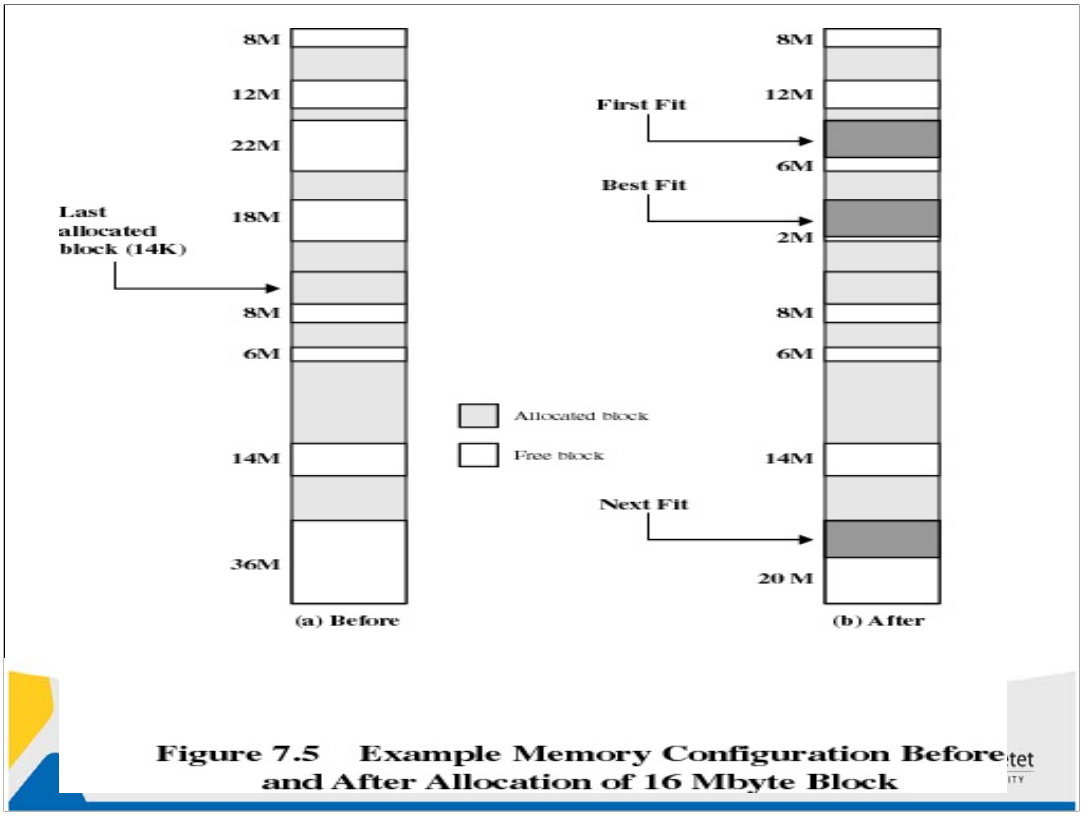
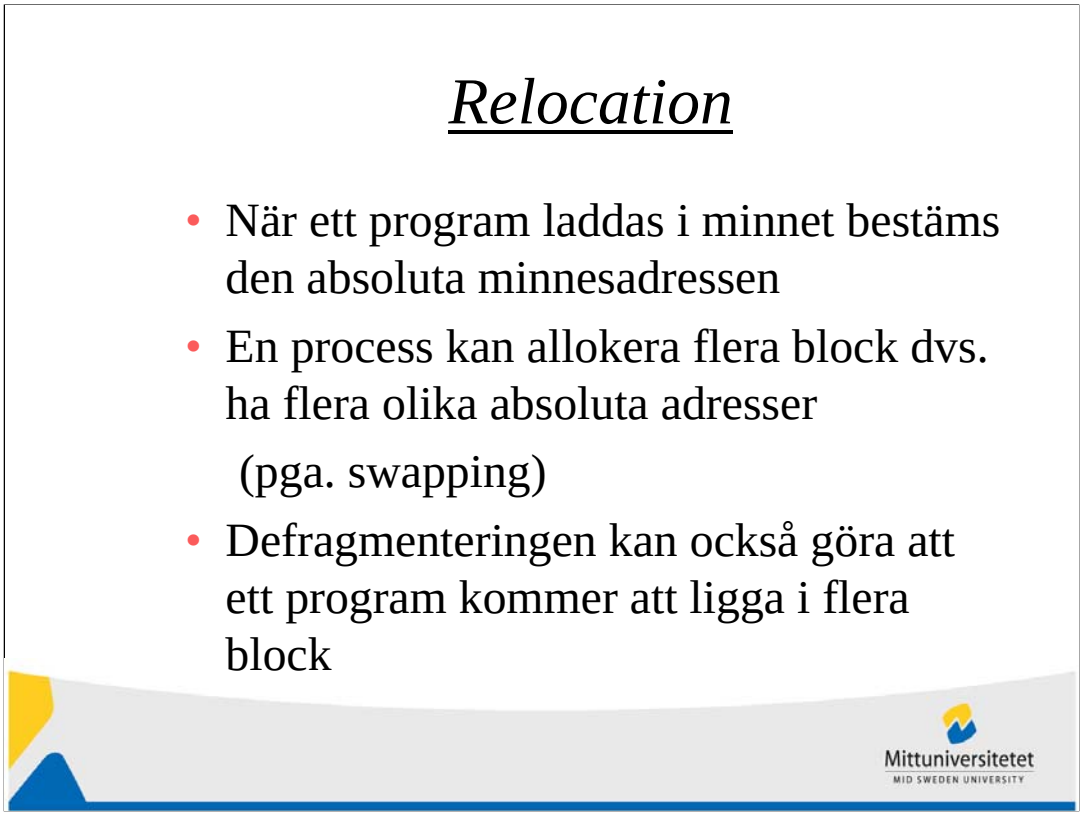


Figure 7.5 Example Memory Configuration Before and After Allocation of 16 Mbyte Block

Bild 20: tillsammans med näst sista stycket på sidan 311 visar hur dessa olika algoritmer skulle placera samma nya process.



Relocation

Då är detta avsnitt desto viktigare. *Bild 24:* med multiprogramming och swappning så kan vi ju inte från gång till gång veta var processens olika delar ligger.

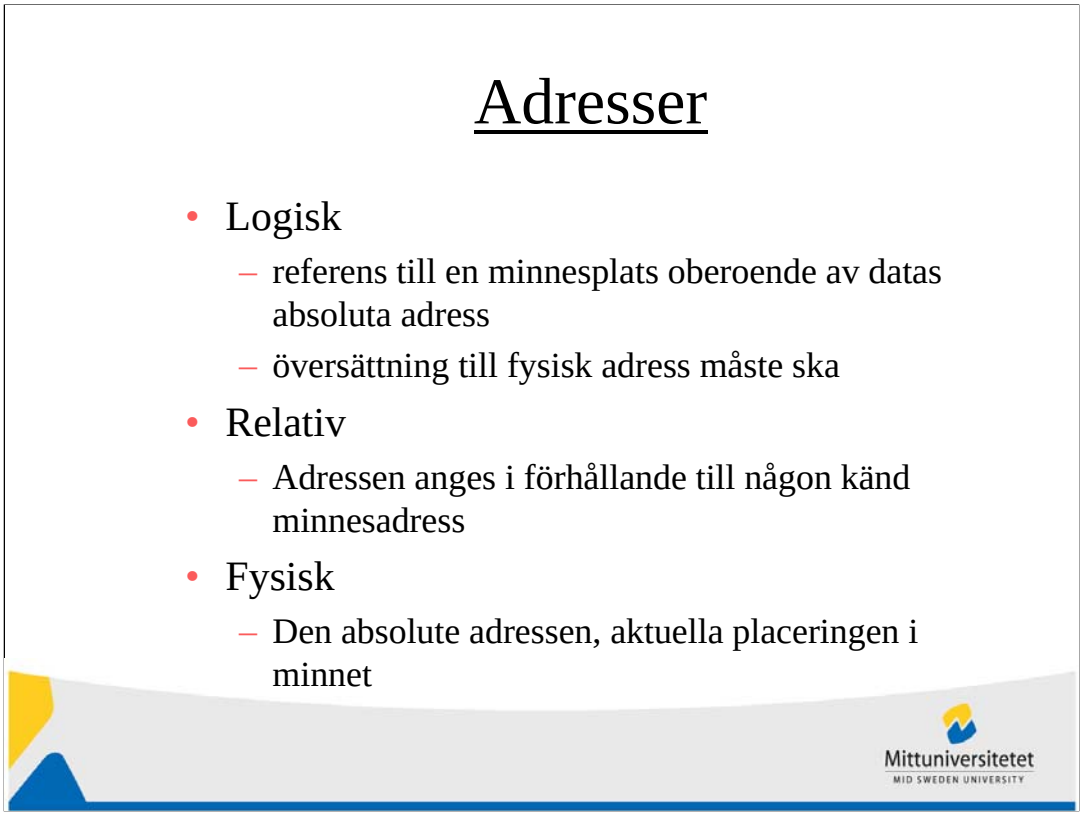


Bild 25: OS måste därför hantera tre olika typer av adresser och översättningar däremellan:

- Logisk adress, den adress som programmet ”tror att” data finns på. Ex A=A+1 ökar värdet på en variabel A som för programmet har en logisk adress. Denna adressreferens måste översättas till en fysisk adress, dvs var i minnet den ligger just nu. Detta gör av MMS.
- Relativ adress, en adress i förhållande till en utgångspunkt. Ex A[2] (C+) adressen till tredje värdet i tabellen A, som har en startadress.
- Fysisk adress, den absoluta adressen i minnet.

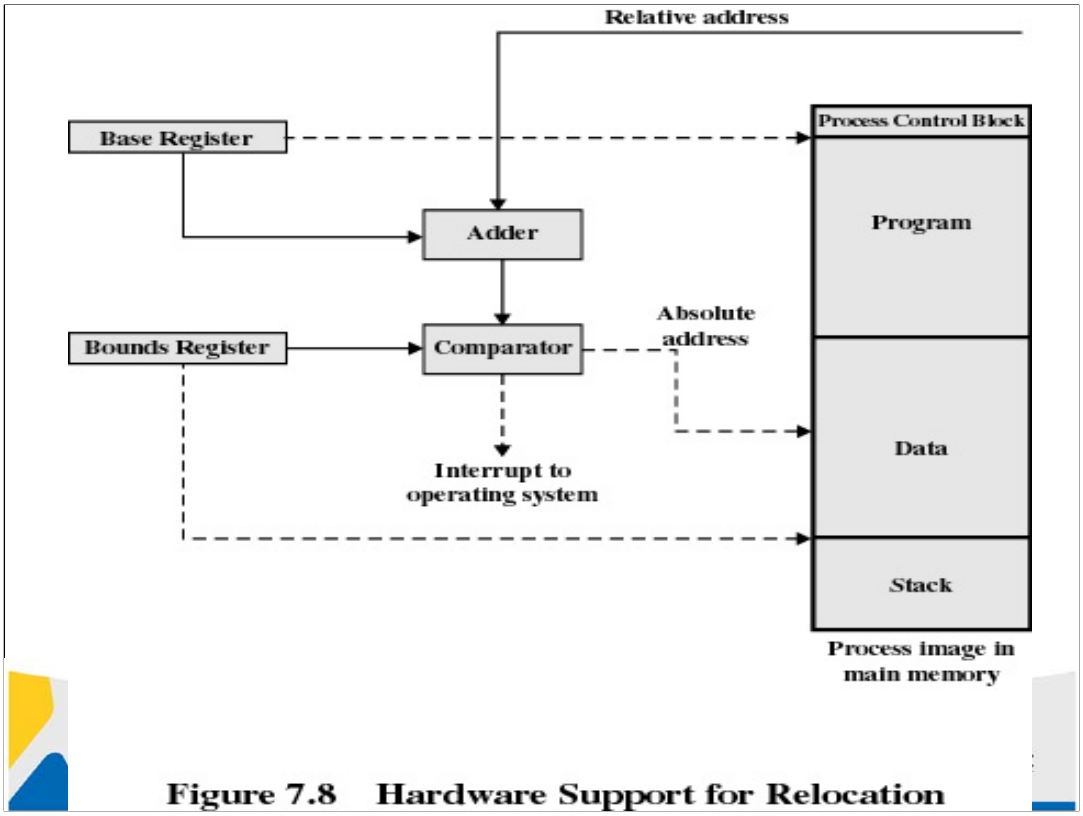


Figure 7.8 Hardware Support for Relocation

Bild 25-27: Viktigt att förstå allt, det blir värre sedan.

Base register innehåller en pekare till programmets faktiska/fysiska startadress.

*Bounds register*används för att kontrollera att processen håller sig inom sina gränser (data protection). Om en adressreferens skulle gå utanför gränsen så skickas ett interrupt som avbryter processen.

Relative address är en referens, som här troligen kommer från processen själv (pilen kunde ha startat någonstans i program). Med en enkel addition fås den fysiska adressen (absolute address). Vi ser här det hårdvarustöd som gör att vi kan garantera *data protection*.

MEN hittills har vi förutsatt att en process får plats i en partition. Däremot visste vi inte i vilken.

Register Som Används Under Exekveringen

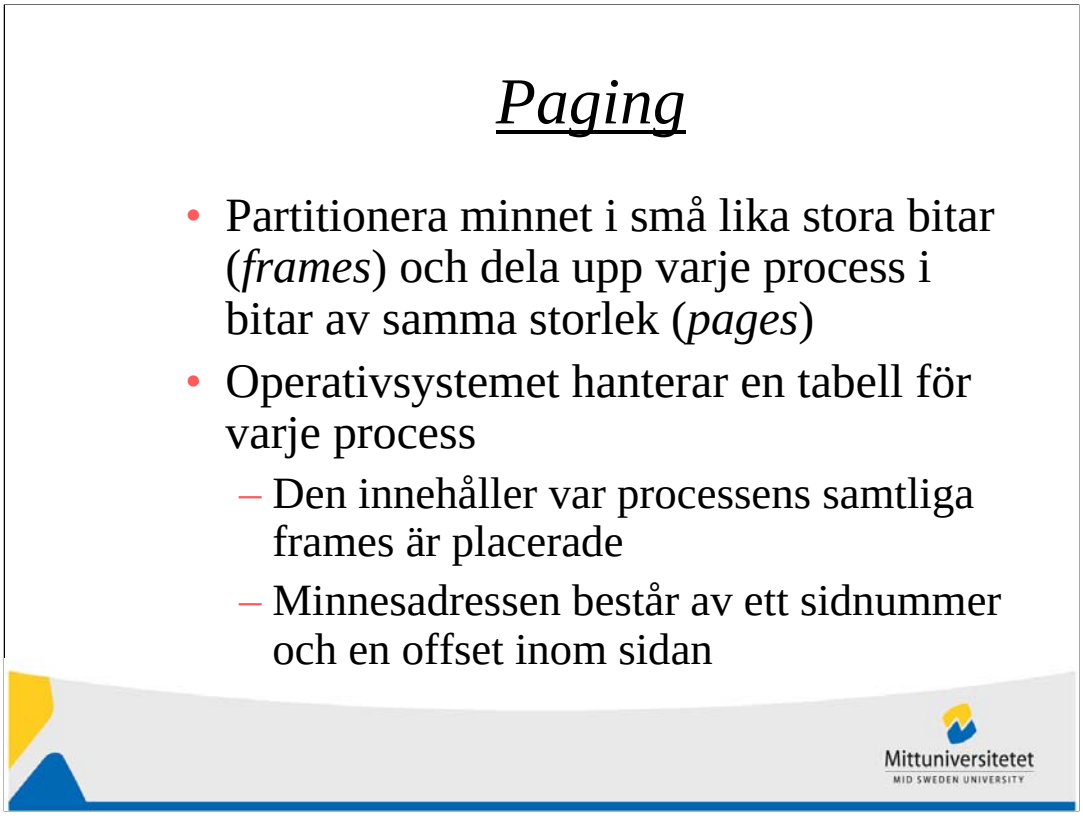
- *Base register* (basadress)
  - Processens startadress
- *Bounds register* (högsta adress)
  - processens sista adress
- Dess värden sätts när processen laddas eller swappas in igen

Mittuniversitetet
MID SWEDEN UNIVERSITY

Register Som Används Under Exekveringen

- Den absoluta adressen fås genom att värdet i *base register* adderas till en relativ adress
- Resultatet jämförs med värdena i *bounds register*
- Om de inte ligger innanför gränserna genereras ett interrupt som operativsystemet får hantera

Mittuniversitetet
MID SWEDEN UNIVERSITY



Paging

Partitionering är inte speciellt effektivt. Vi börjar om och gör på ett annat sätt. Vi delar upp hela minnet i mycket mindre bitar *frames* (ramar). (Denna uppdelning sköter OS helt osynligt för programmeraren (storlek på partition är c:a Mbyte, för Frame c:a Kbyte). Vi (OS) delar sedan även upp processen i lika stora bitar kallas *pages* (sidor). En page kommer då precis att passa i en frame. Vi ser även till att sekundärminnet anpassas till denna storlek.

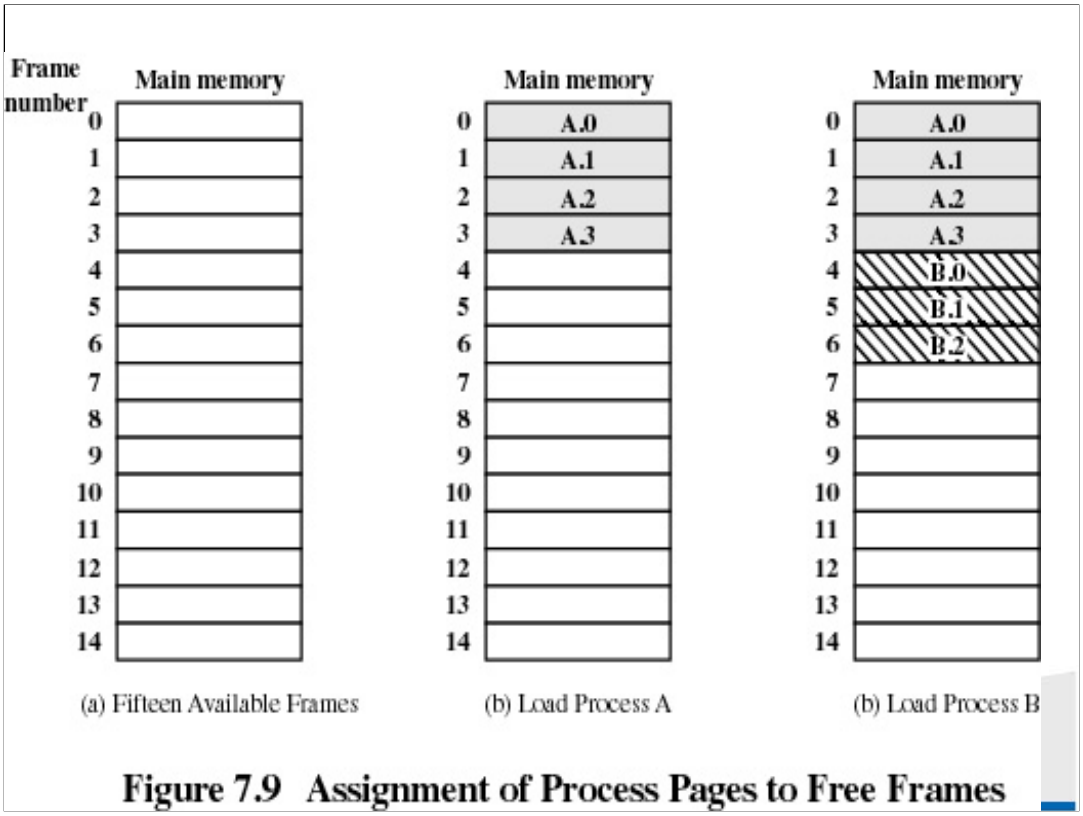


Figure 7.9 Assignment of Process Pages to Free Frames

Bild 30: visar hur minnet består av 15 frames som fylls med processer A,B,C,D. Varje process består av flera sidor och behöver använda flera frames. Processerna fyller det utrymme de behöver.

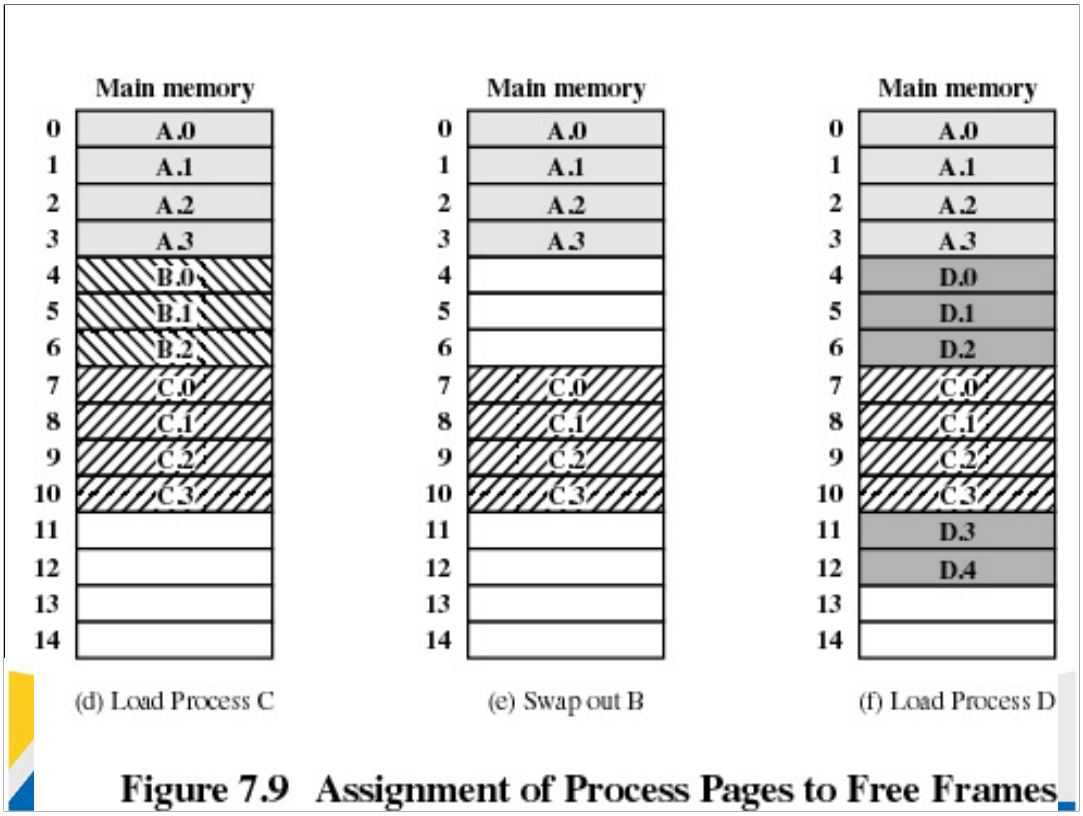


Figure 7.9 Assignment of Process Pages to Free Frames

Bild 31-32: Efterhand, som fler processer tas in och blir klara, blir det inte lika välordnat längre. För att hålla reda på vilken process som för ögonblicket har vilken page i vilken frame behövs en sidtabell (pagetable) för varje process. Det behövs också en tabell med fria frames.

Hur i hela friden klarar vi nu av adressreferenser och dataskydd? Jo, varje logisk adress består av ett sidnummer och en offset (relativ adress) inom sidan. Ligger min variabel A på sidan 3 med offset 12 så översätts det m h a sidtabellen till en frame (fysisk adress) och offset är ju nu inom framen.

Detta system liknar ju fasta partitioner, skillnaden är att framen är så mycket mindre. Dessutom behöver de inte ligga i en följd.

Exempel På Sidtabeller

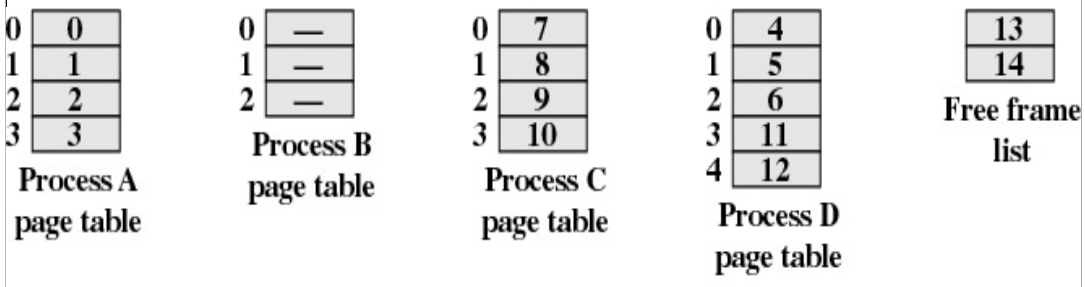


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

Mittuniversitetet
MID SWEDEN UNIVERSITY

Segmentering

- Segment som hör till ett program behöver inte ha samma längd
- Det finns en maximal segmentlängd
- Adressen består av ett segmentnummer och en offset inom segmentet
- Eftersom segmenten kan vara olika stora liknar det dynamisk partitionering

Mittuniversitetet
MID SWEDEN UNIVERSITY

Logiska adresser

I programmet består en logisk adress av ett page-nummer och en offset. Hur kan OS dela upp programmet i sidor i efterhand? Exemplet i boken mitt på sidan 319 till sidan 321 visar hur det går till.

Vi bestämmer oss för en sidstorlek på 1K=1024 bytes. För att kunna adressera dessa 1024 adresser behövs 10 bitar (kommer ni ihåg att 2^10=1024) Med en adresstorlek på 16 bitar blir det då kvar 6 bitar för sidnummer. De 10 minst signifikanta bitarna visar offset och resterande 6 mest signifikanta visar sidan.

Exemplets adress 1502 blir ju binärt 000001-0111011110 (jag har satt ett – mellan sida och offset). Det innebär att adressen 1502 finns på page=1 (000001) med en offset=478 (0111011110).

Genom att välja en sidstorlek som är på formen x^2 förenklas uppdelningen i sidor.

Figure 7.11 sidan 320 i boken, visar i fallet (a) hur hela programmet ligger i en partition. Den logiska adressens offset gäller direkt inom partitionen, eftersom programmets början sammanfaller med partitionens början.

I (b) så används paging och den logiska adressen blir en offset inom en sida. Hela programmet fick plats i tre sidor.

I(c) Är programmet segmenterat (kanske kod och data) man har här valt att använda 12 bitar för segmentstorlek och resterande 4 bitar för segmentnummer. Adressen består av segmentnummer och offset inom segmentet.

Figure 7.12 sidan 321 visar hur enkel översättningen från logisk till fysisk adress blir med ren paging.

• • Maska fram de sex bitarna för page-nummer

• • Slå upp i processens pagetabell i vilken fram sidan ligger

• • Byt ut pagenummret i den logiska adressen mot framenummret -> fysisk adress.

Med segmentering blir det bara lite bökigare. Processens segmentabell innehåller segmentens längd och startadress.

• • Maska fram segmentnummer, de fyra mest signifikanta bitarna.

• • Använd detta segmentnummer för att ur tabellen få den fysiska startadressen.

• • Jämför offsetadressen (resterande 12 bitar) med segmentets längd, för att kontrollera att referensen är giltig (inom segmentet)

• • Addera den fysiska startadressen och offset -> fysisk adress.

Segmentation

Bild 33: Segmentering av programmet är däremot upp till programmeraren. (Till skillnad från paging som sköttes helt osynligt av OS). När man skriver ett program kan det vara lämpligt att dela upp det i flera delar, utifrån logisk synpunkt. Mest uppenbart i kod och data, men man kan ju också skapa moduler på objektbas.

Dessa segment laddas var för sig i minnet och kan varar olika stora. Adressen består nu av ett segment och offset inom detta. Det blir nu mera komplext att kontrollera att referenser sker inom det egna segmentet. Man får ha en segmenttabell som laddas i ett register så att hårdvaran kan kontrollera referenserna fortlöpande.

Om ni förstår allt nu så kommer vi att göra det ännu intressantare i nästa kapitel. Vi ska blanda ihop allt och lägga till virtuellt minne. Dvs OS ska hantera minnesreferenser som inte ens finns fysiskt.